SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4912

Ručno određivanje lažnih preklapanja koja nastaju pri sastavljanju genoma

Filip Floreani

Zagreb, lipanj 2017.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Kako biste uklonili ovu stranicu, obrišite naredbu `\izvornik`.

Zahvaljujem se mentoru Mili Šikiću i Robertu Vaseru na svoj pomoći prilikom izrade ovog rada.

Sadržaj

1. Uvod

Bioinformatika je interdisciplinarno znanstveno područje koje obuhvaća računarsku znanost, statistiku i biologiju te inženjerskim pristupom razvija programsku podršku i metode za analizu bioloških podataka. Kao pojam, bioinformatika se pojavila još sedamdesetih godina prošlog stoljeća, no originalno se odnosio isključivo na proučavanje informacijskih procesa u biotičkim sustavima. Razvojem tehnologije, područje bioinformatike širi se te danas obuhvaća discipline poput analize sekvenci genoma raznovrsnih organizama, proučavanja organizacije stanica, generiranja i analize strukture proteina, itd.

Područja sekvenciranja i analize genoma doživjela su veliku popularizaciju zahvaljujući drastičnom smanjenju cijene i porastu brzine procesiranja podataka, što je dovelo do stvaranje usluga koje nude mogućnost analize vlastitog DN[illegible]o narudžbi. Uz moderne sekvencijske uređaje 3. generacije[1], cijena je najviše kontrolirana efikasnošću algoritama i sustava za sastavljanje genoma. Jedan od predloženih sustava kojim bi se povećala preciznost i efikasnost sastavljanja genoma je sustav temeljen na algoritmima strojnog učenja. Uz pomoć takvih algoritama, sustav dobiva na preciznosti svaki put kada obradi neki skup podataka tj. na temelju obrađenih podataka automatizirano vrši interne modifikacije kako bi ispravio nastale pogreške.

Problem pri izradi takvih sustava, a ujedno i problem koji ovaj rad nastoji riješiti, je generiranje dovoljno velike količine podataka za treniranje algoritama. Sustav razvijen u sklopu ovog rada nastoji pojednostaviti proces sortiranja grafova po predefiniranim kategorijama, kao i proces označavanja neispravnih regija grafova. Stvaranjem više kategorija grafova omogućuje se lakše praćenje napretka treniranja algoritma, a time i postizanje veće preciznosti i učinkovitosti takvog sustava, dok označavanje neispravnih regija grafova daje primjere kojima se može ispitati reakcija sustava na neispravne podatke.

Drugo poglavlje dat će kratki uvod u proces sastavljanja genoma, problematiku koja se u njemu javlja te opisuje sam postupak sastavljanja. Detaljno će se objasniti i

[1] Uređaji za sekvenciranje koji čitaju nukleotidne sljedove na razini molekule

način stvaranja lažnih i kimernih preklapanja te posljedice koje uslijed njih nastaju.

Treće poglavlje opisat će korištene formate i oblik podataka na kojima se temelji razvijeni sustav.

Četvrto poglavlje sadržavat će opis izvedbe programskog rješenja sustava, uz informacije o arhitekturi korisničke (mobilne) i poslužiteljske strane te toka podataka kroz sustav.

Peto poglavlje prikazat će rezultate ispitivanja sustava na testnim podacima.

Šesto poglavlje iznijet će kratki zaključak o razvijenom sustavu te iznijeti prijedloge za daljnji razvoj.

2. Sastavljanje genoma

Sastavljanje genoma može se podijeliti u dva različita pristupa - *de novo* metoda i metoda mapiranja.

Metoda mapiranja temelji se na postojanju referentnog genoma koji je nastao sastavljanjem očitanja dobivenih od više jedinki iste vrste. Pri procesu sastavljanja genoma, uspoređivanje i poravnavanje segmenata vrši se prema referentnom genomu, koji je sličan, ali ne nužno i identičan genomu koji se trenutno sastavlja. Ova metoda može se slikovito prikazati kao sastavljanje velikog kompleta puzzli u kojemu je većina puzzli međusobno slična ili identična, a neki od komadića greškom čak i nedostaju, no postoji vrlo sličan komplet koji je već sastavljen te se pomoću njega može sastaviti i ovaj komplet.

Pri sastavljanju genoma *de novo* metodom, ne postoji referentni genom. To čini ovu metodu mnogo kompliciranijom i težom. U kontekstu vremenske i memorijske složenosti, *de novo* metoda je za red veličine složenija od metode mapiranja. Slikovito, ova metoda može se ponovno zamisliti kao slaganje puzzli, no ovaj put uopće nije poznato kako treba izgledati konačna slika.

2.1. Ponavljajuća i kimerna očitanja

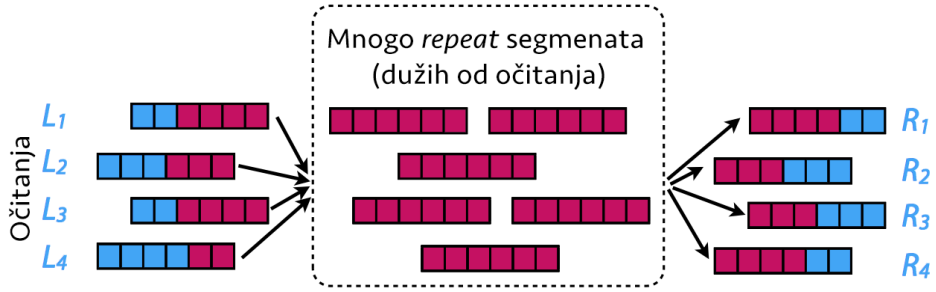
Pri sekvenciranju genoma nekim od modernih sekvencijskih uređaja, potrebno je odlučiti cilja li se na veću duljinu očitanih segmenata ili veću preciznost očitanja.

Naime, zbog male duljine segmenata, poslije nastaje problem pri sastavljanju, uzrokovan mnoštvom ponavljajućih (engl. *rep*[comment] očitanja tj. identičnih ili vrlo sličnih sljedova unutar genoma. Upravo takva očitanja čine oko 50% ljudskog genoma, a pojavljuju se i kod mnogih drugih životinjskih i biljnih vrsta. Ponavljajuće regije nastaju kao posljedica djelovanja mnogih bioloških mehanizama te se u genomu pojavljuju u različitim oblicima i duljinama, od sljedova sastavljenih od tek 1 ili 2 nukleinske baze pa sve do onih sastavljenih od nekoliko milijuna baza.

Iz računarske perspektive, ponavljajuća očitanja uzrokuju dvosmislenosti pri procesima poravnavanja i sastavljanja genoma. Ako ne postoji niti jedno dovoljno dugačko

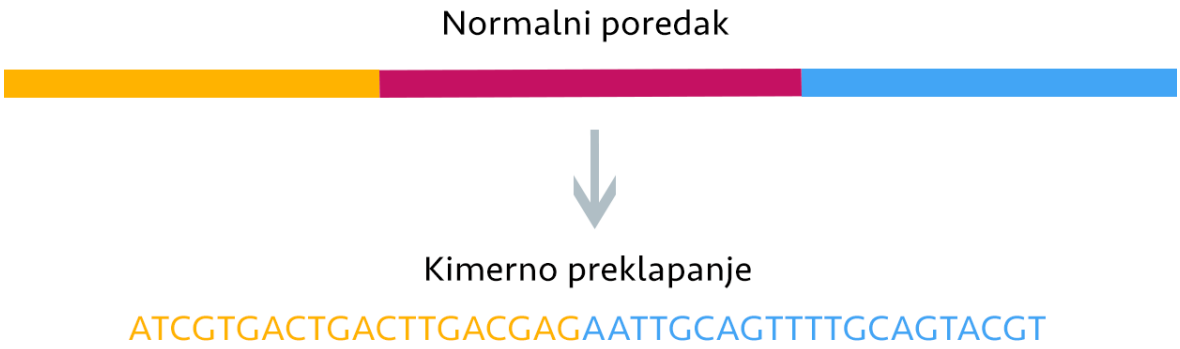
očitanje koje može premostiti *repeat*, sustav ne može garantirati koje očitanje je potrebno staviti na početak, a koje na kraj tog ponavljajućeg očitanja.

Zanemarivanje ponavljajućih regija pri sastavljanju također nije opcija, jer time kasnije može doći do pojave grešaka ili neispravne interpretacije rezultata, ali i zanemarivanja nekih važnih bioloških fenomena.



Slika 2.1: Grafički prikaz ponavljajućeg očitanja

Uslijed smanjene preciznosti očitanja zbog nesavršenosti sustava za sekvenciranje, može doći do pojave lažno pozitivnih očitanja, koja su u stvarnosti međusobno vrlo udaljena. Takva očitanja nazivaju se *kimerna*[2] očitanja. Pojava kimernih očitanja ima nepovoljan utjecaj na konačni rezultat te je takva očitanja nužno razriješiti prije sastavljanja.



Slika 2.2: Grafički prikaz *kimernog* očitanja

[2]Kimera (grč. Himera) – biološka jedinka građena od stanica više različitih zigota

3. Podatkovna struktura

3.1. PAF format

PAF (engl. *Pairwise mapping format*) format podataka definira sadržaj datoteka koje opisuju približna područja preklapanja dviju sekvenci. Svaka datoteka sastoji se od niza redaka, a u svakome od njih nalazi se 12 predefiniranih stupaca međusobno odvojenih znakom `TAB`. Pregled sadržaja stupaca dan je u sljedećoj tablici:

Stupac	Tip	Opis
1	string	Naziv sekvence upita
2	int	Duljina sekvence upita
3	int	Početak upita
4	int	Kraj upita
5	char	Relativni slijed: "+" ili "-"
6	string	Naziv ciljane sekvence
7	int	Duljina ciljane sekvence
8	int	Početak ciljane sekvence na originalnom slijedu
9	int	Kraj ciljane sekvence na originalnom slijedu
10	int	Oznaka poklapanja dviju sekvenci
11	int	Ukupni broj poklapanja, promašaja, ubacivanja i brisanja
12	int	Kvaliteta mapiranja (0-255; 255 označuje promašaj)

Tablica 3.1: PAF format podataka

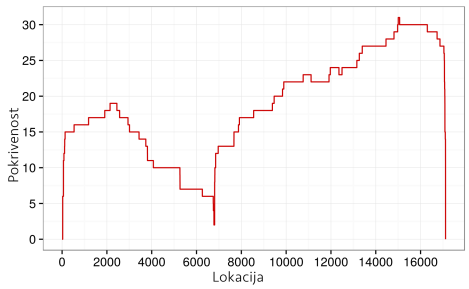
Na kraju svakog retka mogu se nalaziti dodatne vrijednosti oblikovane po principu *ključ – vrijednost*, po uzoru na SAM[3] format podataka.

[3]SAM (engl. *Sequence Alignment/MAP*) format – Tekstualni format za pohranu bioloških sekvenci poravnatih prema referentnoj sekvenci

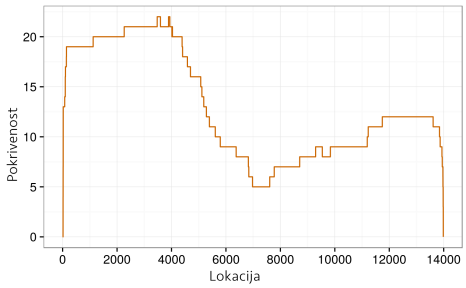
3.2. Grafovi preklapanja

Iako se čini vrlo jednostavan, PAF format teško je razumjeti u tekstualnom obliku. No, obradom tekstualnih podataka iz PAF datoteka, moguće je informacije o preklapanjima sekvenci prikazati u obliku grafa. Svaki graf predočava se u obliku vektora, koji sadrži koordinate njegovih točaka. X koordinata svake točke predstavlja njezinu lokaciju unutar sekvence, dok Y koordinata govori o broju različitih očitanja kojima je ta točka pokrivena.

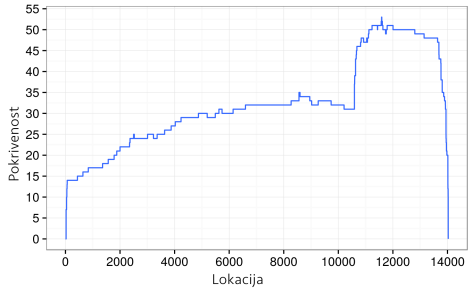
Kroz istraživanje njihovog oblika, uočilo se da postoje 4 karakteristične vrste grafova preklapanja - *repeat*, kimerni, *low quality* i regularni grafovi. Kimerni grafovi mogu se prepoznati po izraženom "šiljku", koji predstavlja neispravno spajanje očitanja. *Repeat* grafove karakterizira jedna (a ponekad i više) regija grafa koje imaju izrazito veliko pozitivno odstupanje pokrivenosti, što je indikator postojanja ponavljajućih očitanja. *Low quality* grafovi prepoznatljivi su po manjim vrijednostima pokrivenosti. Posljednja kategorija su regularni grafovi. Oni nemaju nikakvih izraženih odstupanja u pokrivenosti, a vrijednosti pokrivenosti najčešće su u intervalu između 30 i 40.



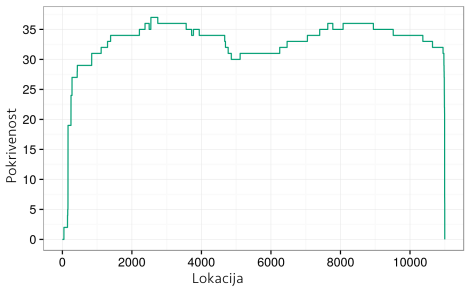
(a) Graf kimernog preklapanja



(b) Graf *low quality* preklapanja



(c) Graf *repeat* preklapanja



(d) Graf regularnog preklapanja

Slika 3.1: Prikaz karakterističnih vrsta grafova preklapanja

4. Programsko rješenje

Programsko rješenje sastoji se od nativne mobilne aplikacije za Android sustave[4] te udaljenog poslužitelja[5], a sustav je temeljen na jednostavnoj klijent – poslužitelj arhitekturi. Ova arhitektura pogodna je za izradu sustava koji procesiraju velike količine podataka, kao i za sustave koji zahtijevaju jednostavno skaliranje snage i resursa.

Klijent i poslužitelj oboje imaju troslojnu internu arhitekturu. Baza podataka na klijentskoj strani nalazi se na mobilnom uređaju, dok se baza podataka na poslužiteljskoj strani nalazi na odvojenom poslužitelju[6].

4.1. Sustav Android i vanjske biblioteke

Mobilna platforma i Android operacijski sustav odabrani su kao dio rješenja jer omogućuju najbolju interakciju s korisnikom za ovu vrstu obrade podataka, a zbog svoje raširenosti, Android omogućuje i jednostavan pristup velikom broju krajnjih korisnika.

Temeljni je način razvoja nativnih Android aplikacija uporabom Android SDK (engl. *Software Development Kit*). Android SDK paket je raznovrsnih alata koji olakšavaju razvoj i testiranje aplikacija, kao i dokumentiranje te uklanjanje programskih grešaka (engl. *bug*). U paketu se, osim primjera programskog koda i biblioteka za razvoj, nalazi i emulator Android uređaja kojim se aplikacija može testirati u kontroliranim uvjetima, kao i ADB (engl. *Android Debug Bridge*), kojim se ostvaruje komunikacija između računala i mobilnih uređaja prilikom testiranja razvijenog proizvoda. Primarni programski jezici pri razvoju su Java i XML[7]. Java se koristi za izradu pozadinskog dijela aplikacije - modela sustava, upravljača i komunikacijskih servisa, dok se XML koristi za definiranje izgleda aplikacije - rasporeda komponenata

[4]https://github.com/ffloreani/Overlapr

[5]https://github.com/ffloreani/Overlapr-server

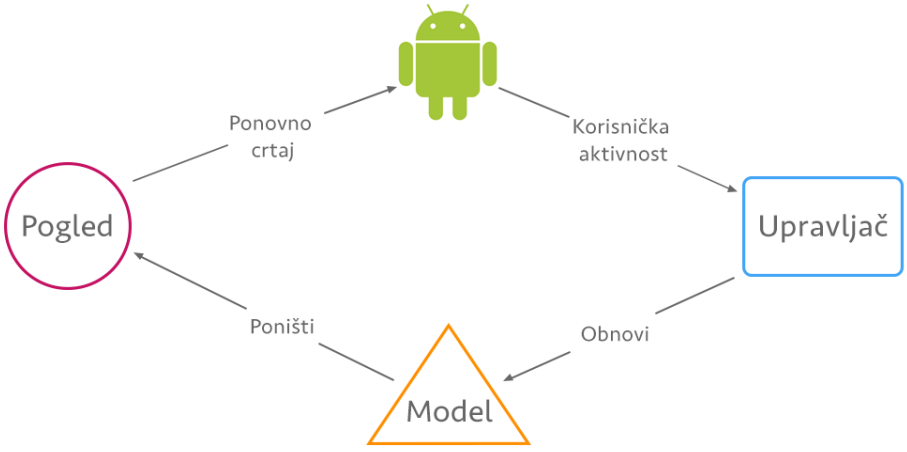
[6]Razdvajanjem baze podataka i aplikacijskog sloja na različite poslužitelje lako je horizontalno skalirati aplikacijski poslužitelj, a poboljšanja su vidljiva i u pogledu sigurnosti i performansi

[7]Od nedavno, nativne Android aplikacije mogu se izrađivati i u programskim jezicima Go i Kotlin

na ekranu, animacija, sadržaja izbornika te nekih predefiniranih vrijednosti važnih za funkcioniranje sustava u cjelini.

4.1.1. Arhitektura Android aplikacije

Arhitektura Android aplikacije razvijene u ovom radu temelji se na MVC pristupu (engl. *Model – View – Controller*). MVC arhitektura odvaja modele od pogleda, koji time postaju prenosivi i, barem teoretski, jednostavniji za održavanje i modifikaciju. Između modela i pogleda stoji upravljač koji je odgovoran za njihovu pravilnu komunikaciju.



Slika 4.1: Grafički prikaz MVC arhitekture

Sloj modela

Sloj modela sadrži podatke, stanja i kompletnu poslovnu logiku sustava, a predstavljen je Java razredima koji sadrže privatne parametre kojima se pristupa uz pomoć "getter"[8]/"setter"[9] metoda. Zbog korištenja modela za komunikaciju s Realm bazom podataka, svi modeli izvedeni su iz `RealmObject` razreda, koji u sebi enkapsulira sve potrebne metode za uspješan rad s bazom podataka. Iz istog razloga, svaki model sadrži i jedinstveni UUID[10] identifikator. Za definiranje statičkih nepromjenjivih vrijednosti kojima se definiraju prethodno spomenute vrste karakterističnih grafova, koristi se enumeracija te se ona također može smatrati dijelom aplikacijskog modela.

[8] Getter – Metoda za dohvat vrijednosti parametra
[9] Setter – Metoda za postavljanje vrijednosti parametra
[10] UUID (engl. *Universally Unique Identifier*) – 128-bitna vrijednost generirana po RFC 4122

Upravljački sloj

U Android MVC arhitekturi, upravljački sloj građen je od tzv. `Activity` razreda. Kao što im i ime kaže, ovi razredi odgovorni su za sve aktivnosti koje se odvijaju unutar aplikacije. Životni ciklus svake Android aplikacije koncipiran je upravo oko `Activity`-a. Njihova definicija vrši se u internom aplikacijskom manifestu, kojeg operacijski sustav analizira pri instalaciji aplikacije te tako zna kada i kako pravilno aktivirati potrebni `Activity`. Svaki Android `Activity` odgovoran je za pravilno pokretanje i zaustavljanje prikaza pogleda, ispunjavanje pogleda s podacima preuzetim iz modela tj. baze podataka te pozivanje različitih sustavskih komponenata poput servisa i asinkronih zadataka[11], koji se također nalaze u upravljačkom sloju, a njima se omogućuje komunikacija aplikacije s vanjskim procesima i sučeljima.

Sve aplikacije na nekom Android uređaju pri svom pokretanju od sustava dobivaju jednu dretvu za izvođenje, na kojoj se vrše sve operacije korisničkog sučelja te se ona stoga naziva glavna dretva ili dretva korisničkog sučelja (engl. *UI thread*). Za dobro korisničko iskustvo, od iznimne je važnosti održati fluidnost korisničkog sučelja tj. ne dozvoliti da ga neki dugotrajan proces blokira. U suprotnom, aplikacija prestaje reagirati na korisničke naredbe te ju operacijski sustav automatski prekida i gasi. Zbog svega navedenog, nužno je za sve dugotrajne procese, poput komunikacije s udaljenim poslužiteljima ili čitanja/pisanja datoteka koje se nalaze u sustavskoj memoriji, koristiti servise i asinkrone zadatke kako se ne bi narušila stabilnost aplikacije.

Sloj pogleda

Sloj pogleda odgovoran je za reprezentaciju modela. Komponente sloja pogleda su razredi `View` i `ViewGroup` te njihovi podrazredi. Svaki od tih razreda povezan je s jednim XML elementom koji definira raspored, orijentaciju, dimenzije i oblik jedne vizualne komponente ili grupe više njih. Jedan od zadataka razreda u sloju pogleda je pobrinuti se za prikaz sadržaja primljenog od upravljača, pritom vodeći računa o zahtjevima postavljenim u XML definiciji pripadajuće komponente. Druga je zadaća ovih razreda proslijediti obavijest upravljaču o svim korisničkim aktivnostima nad pogledom. Iako MVC arhitektura nastoji to izbjeći, pri prijenosu informacija prema upravljačkom sloju ponekad dolazi do miješanja upravljačkog sloja i sloja pogleda kako bi se informacija uspješno prenijela i obradila. Ovo je slučaj, primjerice, pri prikazivanju složenih animacija, kada je potrebno primiti pokret koji je korisnik izvršio na ekranu, procesirati primljeni pokret i potom izvršiti i prikazati pripadajuću animaciju.

[11] Asinkroni zadatak (engl. asynchronous task) – Komponenta koja odsječak programskog koda izvodi u pozadinskoj dretvi, a rezultat dostavlja na dretvu korisničkog sučelja

4.1.2. Vanjske biblioteke

Iako moćan, Android SDK nije uvijek dostatan za ispunjavanje svih zahtjeva sustava. U tu svrhu, koriste se dodatne vanjske biblioteke.

Za jednostavnu i automatiziranu izgradnju projekata, Android Studio razvojno okruženje (engl. *Integrated Development Environment*) koristi alat Gradle. Unutar `gradle.build` datoteka (u svakom projektu postoje minimalno dvije), nalaze se sve postavke potrebne za uspješnu pretvorbu programskog koda u funkcionalnu mobilnu aplikaciju. Korištenje Gradle alata je ujedno i najjednostavniji način za dodavanje vanjskih biblioteka u Android projekte.

Glavne su biblioteke u ovom projektu MPAndroidChart, Realm Mobile Database i SwipeableCardStack. Ostale korištene biblioteke uključuju NoNonsense-FilePicker i OkHttp3.

MPAndroidChart

MPAndroidChart svestrana je biblioteka namijenjena kreiranju i prikazivanju grafova različitih oblika. Podržava više različitih vrsta grafova, ima visok stupanj individualizacije prikaza, opciju povećavanja prikaza, označavanja regija te mogućnost prikaza podataka direktno iz Realm baze podataka. Koristi se za prikaz grafova prilikom označavanja regija, kao i za prikazivanje grafova na kartama prilikom sortiranja.

Realm Mobile Database

Realm Mobile Database (RMD) objektna je baza podataka dizajnirana za korištenje upravo na mobilnim uređajima. U usporedbi s klasičnom SQL bazom podataka, ona pohranjuje nativne aplikacijske objekte, što pojednostavljuje postupak osvježavanja podataka unutar baze. Umjesto dohvaćanja podatka, kopiranja u podatkovnu memoriju, izmjene sadržaja i ponovnog unosa u bazu podataka, RMD nudi pomoćne metode kojima se objektu unutar baze može direktno pristupiti i mijenjati sve podatke koje on sadrži, što baza automatski dojavljuje svim `Realm` instancama unutar sustava. Za razliku od SQL baza podataka, RMD koristi princip lijenog dohvata podataka (engl. *lazy loading*). Time se smanjuje opterećenje priručne podatkovne memorije, jer se podatci iz baze dohvaćaju tek u trenutku kada su potrebni. Također, pošto se u bazu objekti pohranjuju direktno, sloj modela se maksimalno pojednostavljuje jer više nema potrebe za izradom specijaliziranih ORM-ova[12], što dovodi do čišćeg programskog koda,

[12]ORM (engl. *Object-relational mapper*) - Razred zadužen za rukovanje s objektima pri njihovoj interakciji s bazom podataka

a smanjuje i vjerojatnost pojave grešaka u sustavu.

SwipeableCardStack

Biblioteka SwipeableCardStack omogućuje kreiranje vizualnih komponenti sličnih špilu karata, uz dodatnu mogućnost uklanjanja karata u više smjerova. Osim XML elementa koji opisuje špil, biblioteka sadrži gotove metode i razrede upravljačkog sloja i sloja pogleda kojima se animiraju i izvršavaju sve aktivnosti nad kartama. Unutar ovog projekta, biblioteka se koristi kao kostur za kontrolu karata unutar špila, dok su specifične funkcionalnosti i izgled dodani naknadno.

Ostale biblioteke

Prethodno spomenute biblioteke čine osnovu cijele aplikacije te bi njen razvoj bez njih bio iznimno kompliciran. No, osim njih, u aplikaciji su korištene još neke biblioteke koje pojednostavljuju dijelove poput ostvarivanja komunikacije između aplikacije i poslužitelja te pretrage podatkovne strukture mobilnog uređaja pri traženju datoteka.

Za izgradnju, odašiljanje i primanje HTTP zahtjeva zadužena je biblioteka OkHttp, koja predstavlja jednostavan HTTP klijent koji smanjuje broj potrebnih koraka za uspostavu i održavanje Internet komunikacije. U sklopu biblioteke implementirana je podrška za obnovu veze uslijed čestih pogrešaka, dok je uspostavljanje komunikacije temeljeno na TLS 1.2 kriptografskom protokolu, uz mogućnost korištenja TLS 1.0 kao rezerve. Zadatak OkHttp-a u ovom projektu je stvaranje `HTTP GET` i `POST` zahtjeva prema predefiniranim API rutama na poslužitelju. Svi zahtjevi implementirani su asinkrono koristeći povratne pozive[13] kojima se omogućuje rukovanje ispravnim odgovorima, ali i pogreškama do kojih je moglo doći prilikom prijenosa ili obrade zahtjeva.

Pretraživanje datotečnog sustava mobilnog uređaja potrebno je prilikom odabira preuzetih datoteka s udaljenog poslužitelja. U tu svrhu, koristi se biblioteka NoNonsense-FilePicker. Jedan od razloga korištenja upravo ove biblioteke je ugrađena filtracija datoteka po ekstenziji, čime se onemogućuje odabir nepodržane datoteke u aplikaciji.

4.2. Udaljeni poslužitelj

Udaljeni poslužitelj u sklopu ovog projekta zadužen je za rukovanje velikim brojem datoteka te primanje i odašiljanje tih datoteka na zahtjev. Sustav poslužitelja temeljen

[13]Povratni poziv (engl. *callback*) – Odsječak programskog koda koji se predaje kao argument drugoj metodi te se poziva u nekom trenutku njenog izvođenja

je na tzv. MEAN programskom složaju[14]. Glavno radno okruženje čini Node.js povezan s MongoDB NoSQL bazom podataka. Na Node.js dodan je razvojni okvir Express.js koji pojednostavljuje razvoj web aplikacija i aplikacijskih programskih sučelja (engl. *Application Programming Interface, API*). Za izvođenje na klijentskoj strani zadužen je razvojni okvir AngularJS.

Node.js okruženje omogućuje izgradnju web poslužitelja kroz razvoj temeljen na događajima (engl. *event-driven programming*). Umjesto klasičnog konkurentnog pristupa, gdje se za svaki zahtjev pokreće zasebna dretva, Node.js sadrži beskonačnu petlju događaja (engl. *event loop*) temeljenu na obrascu promatrača, koja osluškuje događaje te za svaki događaj pokreće pripadajuću metodu. Korištenjem petlje događaja, izbjegava se nastanak zastoja procesa te se time otvara mogućnost izgradnje vrlo skalabilnih sustava.

Vanjske biblioteke većinom su okupljene oko upravljača programskih paketa npm (engl. *node package manager*). Neki od korištenih paketa u ovom projektu su Express.js, mongoose.js i Busboy.

Kao što je na početku poglavlja spomenuto, poslužitelj je temeljen na troslojnoj arhitekturi koja podrazumijeva sloj modela, upravljački sloj i sloj pogleda. Za upravljački sloj i sloj modela zadužen je Node.js, dok se za sloj pogleda i komunikaciju prema upravljačkom sloju brine AngularJS. Pokretanje poslužitelja vrši se aktivacijom skripte `bin/www`. Ona kroz skriptu `app.js` okuplja sve potrebe resurse, aktivira URL rute i postavlja osnovnu kontrolu pogrešaka. Potom vrši inicijalizaciju vrata za komunikaciju s mrežom te na njih postavlja promatrač prometa povezan s postavljenim HTTP poslužiteljem. Nakon uspješnog pokretanja, izvođenje prelazi na Node.js-ov *event loop* te je sustav spreman za prihvaćanje zahtjeva. Po primitku zahtjeva, *event loop* i Express.js pomoću metode i putanje zahtjeva otkrivaju koji je događaj izvršen te se pokreće prikladna metoda. Na svaku takvu metodu dodan je povratni poziv kojim se signalizira kraj obrade zahtjeva.

Kroz funkcionalnost Express.js-a, definirane su tzv. rute (engl. *route*) za svaku vrstu zahtjeva prema poslužitelju. Svaka ruta reprezentira jednu krajnju točku (engl. *endpoint*) razvijenog API-a. Svaka ruta definirana je URL putanjom te metodom HTTP zahtjeva. Također, u sklopu definicije rute postoji i povratni poziv s argumentima `req`[15] i `res`[16]. Argument `req` sadrži sve informacije vezane uz primljeni HTTP zahtjev,

[14]MEAN (engl. *MongoDB Express.js AngularJS Node.js*) – Programski složaj temeljen na jeziku JavaScript, namijenjen za jednostavan i brz razvoj skalabilnih web aplikacija

[15]`req` (engl. *request*) – JavaScript objekt kojim je predstavljen HTTP zahtjev

[16]`res` (engl. *response*) – JavaScript objekt kojim je predstavljen HTTP odgovor

uključujući zaglavlje, tijelo te poslane parametre, dok argument `req` služi za generiranje HTTP odgovora koji će biti poslan natrag klijentu. Ovisno o metodi zahtjeva, u odgovoru se može nalaziti ili sadržaj datoteke ili statusna poruka o (ne)uspješnosti poslanog zahtjeva. Popis svih ruta dostupnih unutar razvijenog poslužitelja prikazan je u sljedećoj tablici.

URL putanja	Metoda	Opis
/api/charts	GET	Dohvaća prvu nepreuzetu parsiranu datoteku
/api/pafs	GET	Vraća popis svih pohranjenih PAF datoteka
/api/pafs	POST	Prima PAF datoteku i započinje njeno parsiranje
/api/results	GET	Vraća popis svih pohranjenih datoteka s rezultatima
/api/results	POST	Prima datoteku s rezultatima procesiranja grafova

Tablica 4.1: Dostupne API rute

Obrada HTTP zahtjeva

Upućivanjem zahtjeva prema bilo kojoj od navedenih ruta, pokreće se izvođenje pripadajućeg upravljača. Upravljač je zadužen za komunikaciju s bazom podataka i izvršavanje zahtjeva.

Za obradu `POST` zahtjeva koji sadržavaju datoteke, koristi se paket Busboy. Busboy koristi promatračke metode `Busboy#on(file, ...)` i `Busboy#on(finish, ...)` koje se izvršavaju po primitku datoteke i po završetku zahtjeva. Prije slanja odgovora o uspješnom primitku datoteke, poziva se metoda `Mongoose.Model#save(function)` koja informacije o novoj datoteci pohranjuje u bazu podataka.

Pri radu s `GET` zahtjevima, prvo je potrebno dohvatiti informacije iz baze podataka, a potom u odgovor zapakirati datoteku i poslati ga klijentu. Za dohvat podataka iz baze koristi se paket mongoose.js, napravljen isključivo za rad s MongoDB bazom podataka. Metodom `Mongoose.Model#findOneAndUpdate(query, update, ...)` pronalazi se prva datoteka koja još nije preuzeta te se njen status preuzimanja postavlja na `true` nakon uspješnog izvršavanja upita. Ukoliko se pronađe datoteka koja odgovara takvom upitu, ona se metodom `res.download(path, filename, ...)` pakira u odgovor, koji će na klijentskoj strani inicirati preuzimanje datoteke.

Bazu podataka na poslužiteljskoj strani čini sustav za baze u oblaku (engl. *cloud database service*) mLab na čijem se poslužitelju nalazi MongoDB NoSQL baza poda-

taka. MongoDB koncipiran je kao dokumentna baza podataka, namijenjena za pohranu polustrukturiranih podataka koji nisu ograničeni shemama.

4.3. Tok podataka

Put kojim prolaze podatci započinje na uređaju koji sadrži PAF datoteke. Za slanje datoteka na udaljeni poslužitelj, koristi se web aplikacija s pripadnim sučeljem. Aplikacija konstruira `HTTP POST` zahtjev u kojeg dodaje predanu datoteku te zahtjev upućuje prema poslužitelju. Prilikom obrade zahtjeva, poslužitelj prati postupak opisan u prethodnom potpoglavlju, no prije završetka obrade i slanja odgovora klijentu, dodatnim sistemskim pozivom pokreće program za parsiranje zaprimljene datoteke.

4.3.1. Parsiranje PAF datoteke

PAF datoteke parsiraju se na udaljenom poslužitelju kako bi se uklonila većina mukotrpnog posla s mobilnih uređaja, koji nemaju dovoljno resursa za glatko obavljanje takvih operacija. Čitav proces parsiranja izdvojen je u zasebni Java program koji je zapakiran u JAR[17] datoteku. Program pri pozivu prima 2 argumenta - apsolutnu putanju do PAF datoteke koju je potrebno parsirati te apsolutnu putanju do datoteke u koju će se spremiti rezultat. Po učitavanju PAF datoteke, svaki njen redak procesira se tako da se iz njega iščitaju informacije iz stupaca 1, 3, 4, 6, 8 i 9 (tablica 3.1). Stupci 1, 3 i 4 čine skup informacija o sekvenci upita, a stupci 6, 8 i 9 skup o ciljanoj sekvenci. Potom se za svaki od skupova u izlaznoj datoteci osvježavaju informacije o pripadajućem grafu.

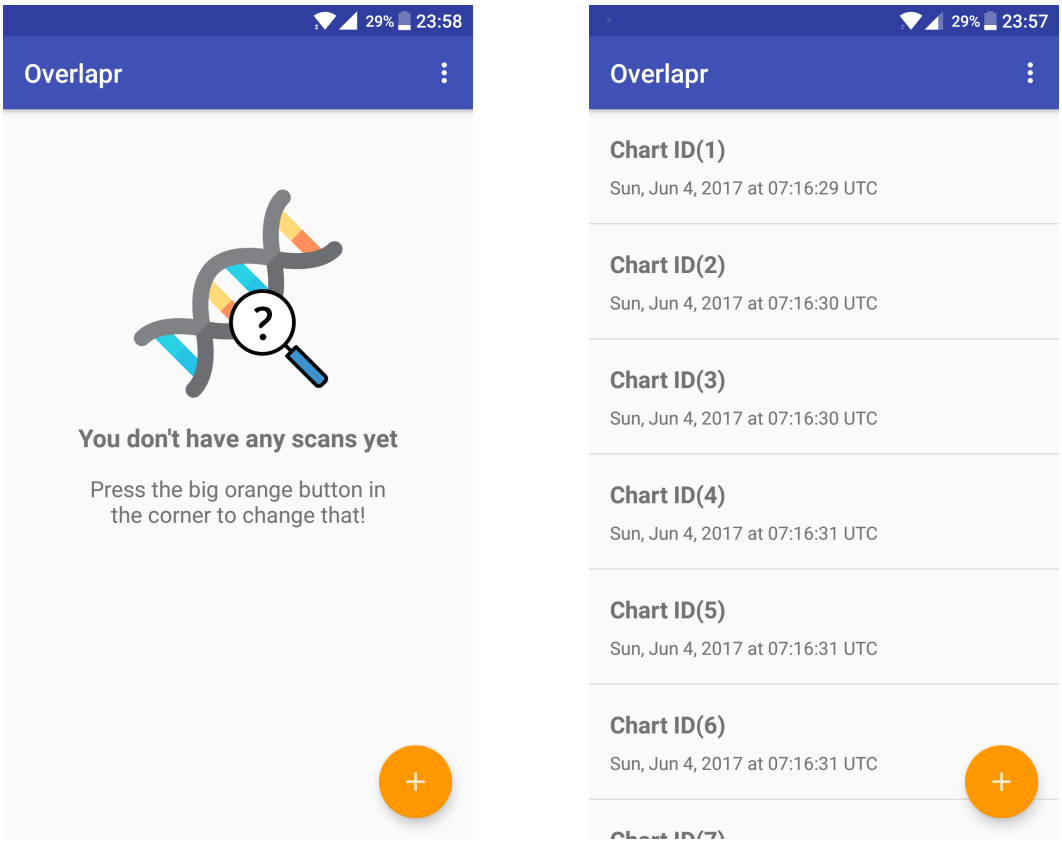
Svaki redak izlazne datoteke predstavlja jedan graf. Na početku retka zapisan je naziv sekvence u obliku `ID(naziv sekvence)`, a potom slijede Y koordinate svake točke danog grafa, međusobno odvojene znakom `TAB`. Ovaj format zapisa odabran je kako bi se maksimalno pojednostavilo kreiranje grafova na mobilnim uređajima, a sam program može se dodatno konfigurirati tako da od jedne PAF datoteke stvori više izlaznih datoteka, čime bi se dobile datoteke s podjednakim brojem grafova. Ovakva dodatna konfigurabilnost stvorena je kako bi se omogućilo slanje manje količine podataka po svakom HTTP zahtjevu, kao i bolja distribucija parsiranih grafova prema više potencijalnih krajnjih korisnika.

Po završetku parsiranja, apsolutne putanje svih izlaznih datoteka zapisuju se u tablicu `Charts` unutar MongoDB baze podataka, te se unutar tablice označuju kao još nepreuzete datoteke.

[17] JAR (engl. *Java ARchive*) – Izvršna Java datoteka koja sadržava sve razrede i biblioteke programa

4.3.2. Graf preklapanja

Parsirane datoteke se slanjem `HTTP GET` zahtjeva, čija je obrada na poslužitelju opisana u prethodnom potpoglavlju, preuzimaju na Android mobilni uređaj. Otvaranjem datoteke unutar aplikacije, pokreće se proces pretvorbe zapisa iz te datoteke u objekte razreda `RealmChartModel` i `RealmPointModel`. Zbog velikog broja grafova unutar jedne preuzete datoteke, pretvorbu je moguće pauzirati, a ponovnim odabirom iste datoteke, pretvorba započinje tamo gdje je posljednji put zaustavljena.



(a) Prikaz praznog popisa **(b)** Popis pripremljenih grafova

Slika 4.2: Izgled korisničkog sučelja za prikaz svih pripremljenih grafova

Svaka instanca razreda `RealmChartModel` ima pripadajući skup točaka koje su predstavljene instancama razreda `RealmPointModel`. Za svaki redak otvorene datoteke stvara se po jedna instanca `RealmChartModel` te joj se dodjeljuje ime jednako prvom zapisu toga retka. `RealmPointModel` objekti dobivaju se indeksiranjem Y koordinata točaka unutar svakog retka, čime je, stoga, određena i X koordinata svake točke. Važno je naglasiti da se pri stvaranju ovih objekata koristila tek svaka šesta točka sadržana u izvornoj datoteci. Razlog ovome je iznimno veliki broj točaka u svakom grafu, prosječno oko 15.000 točaka, što predstavlja vrlo veliku količinu podataka koju je potrebno obraditi prilikom crtanja grafa. Tolika količina podataka stavlja veliko opterećenje na

procesorsku i grafičku jedinicu mobilnog uređaja, što se očituje i u radu aplikacije. Kako bi se taj teret smanjio, sustav procesira smanjeni skup točaka, ali duljina regije grafa s istim vrijednostima dovoljna je da ne dođe do značajnog gubitka informacije.

Nakon povezivanja instance grafa sa svim pripadnim točkama, svi novostvoreni objekti pohranjuju se u lokalnu bazu podataka. Kako je pogled popisa grafova povezan direktno s bazom, pri stvaranju novih grafova dolazi do automatskog osvježavanja popisa kako bi se u njemu predočili dodani grafovi.

Konstrukcija i prikaz grafova

Proces konstruiranja i prikaza jednog grafa počinje prikupljanjem kolekcije objekata `RealmPointModel` iz baze podataka koji su povezani s odabranim grafom te izradom skupa podataka `LineDataSet` koji odgovara tipu linijskog grafa. Pri generiranju skupa `LineDataSet`, osim kolekcije točaka potrebno je predati i informaciju o tome koji će se parametri točaka koristiti na kojoj osi. U ovom koraku definiraju se i neke stilske postavke, poput postavljanja boje linija na grafu, ispune površine ispod grafa, iscrtavanja točaka i slično.

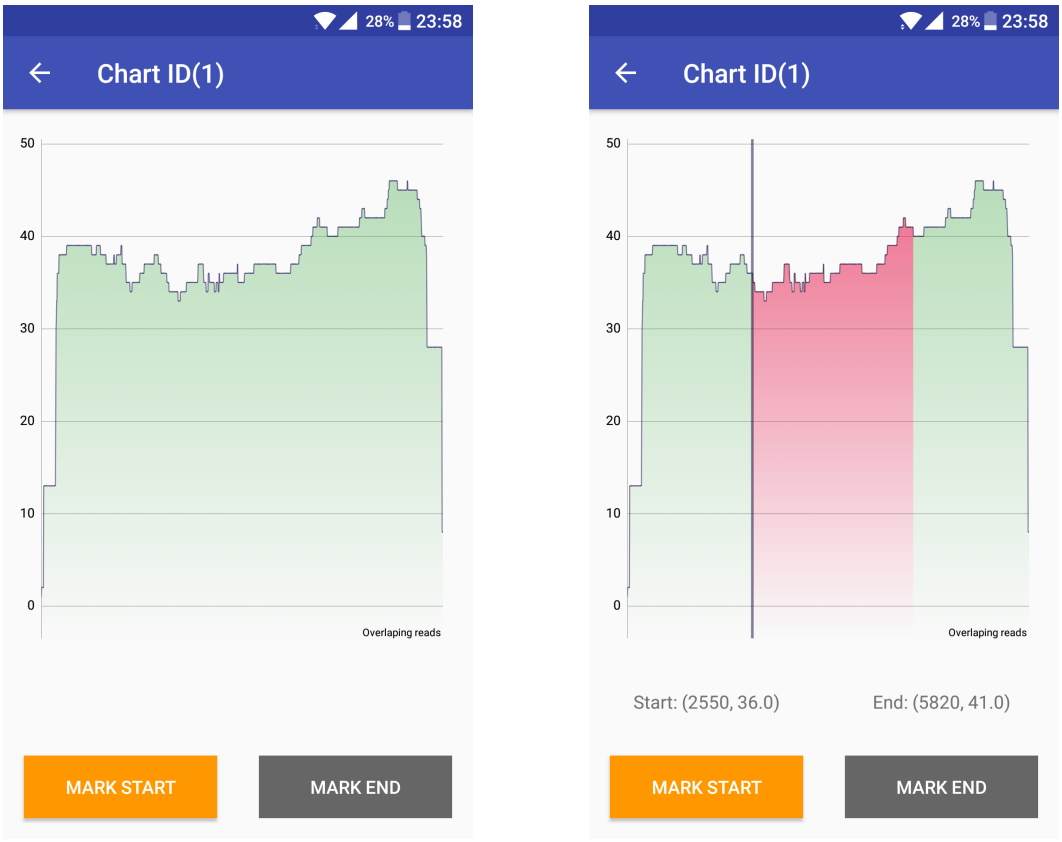
Na temelju kreiranog skupa podataka, stvara se instanca razreda `LineData` koja enkapsulira dosad definirane podatke zajedno s informacijama o mjestu i načinu prikaza koordinatnih osi, legende te pozicioniranju i sadržaju opisne labele. Nakon ispunjavanja svih željenih parametara, objekt `LineData` predaje se objektu pogleda razreda `LineChart` koji potom vrši crtanje grafa i prikazuje ga korisniku.

Označavanje regija

Zadatak korisnika prilikom ručnog određivanja lažnih preklapanja je označiti dio grafa za koji se smatra da predstavlja takvo preklapanje. Koristeći označivač na grafu, potrebno je pozicionirati se na početak regije, a zatim ga i označiti. Drugi korak uključuje pomicanje označivača na kraj regije, kojeg se potom također označi (slika 4.3).

Pri označavanju regije, koristi se objekt razreda `Highlight`, sadržan u prethodno spomenutom objektu `LineChart`, u kojemu se nalazi informacija o koordinatama označene točke. Ako je graf kojeg se označava već označen, tada se stara vrijednost prepisuje s novom vrijednosti. Ovime je omogućeno ispravljanje grešaka koje se mogu dogoditi prilikom označavanja.

Za pohranu informacija o označenoj regiji, izrađen je razred `RealmHighlightModel`, koji je povezan s grafom kojeg se označava te početnom i završnom točkom regije. Pozi-



(a) Neoznačeni graf **(b)** Označeni graf

Slika 4.3: Izgled korisničkog sučelja za označavanje regija

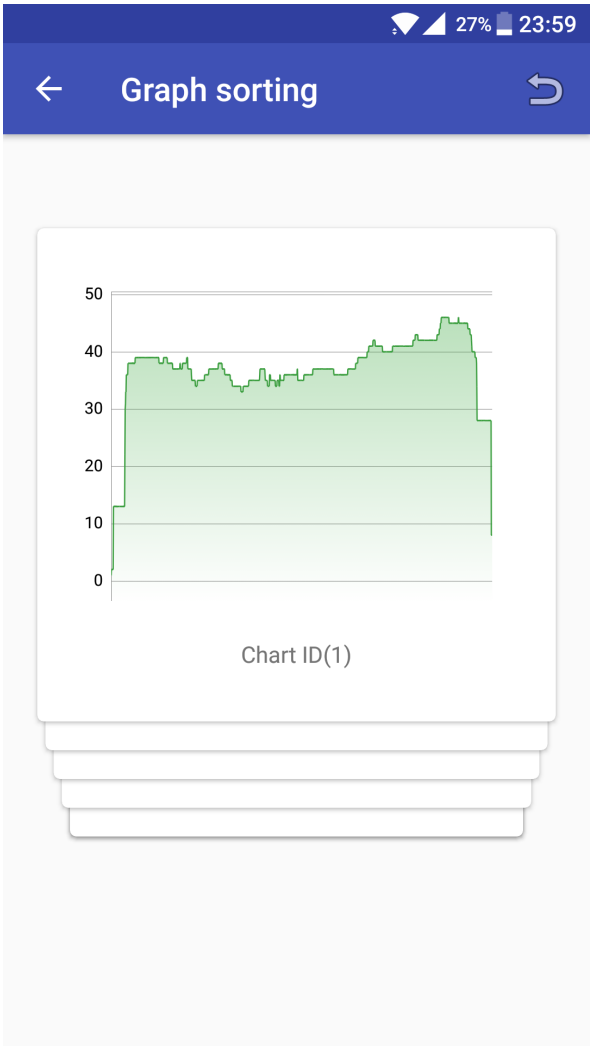
vom asinkrone metode `Realm#executeTransactionAsync(...)`, pokreće se pozadinski proces u kojem dolazi do pohrane objekta u bazu podataka. Rezultat pohranjivanja dojavljuje se kroz jedno od 2 vezana sučelja `RealmTransaction.Callback`. Jedno sučelje koristi se za definiciju radnji uslijed uspješnog zapisivanja, dok se drugo sučelje poziva u slučaju pogreške.

U slučaju kada pohrana prođe bez pogrešaka, originalni objekt razreda `LineDataSet` dijeli se na 3 nova objekta, od kojih jedan sadrži točke iz označene regije grafa, a druga dva sadrže preostale točke. Kako bi se ova promjena uspješno prikazala, potrebno je poništiti objekt `LineChart`, čime će se automatski pokrenuti njegovo ponovno iscrtavanje.

Sortiranje grafova

Sortiranje grafova vrši se po kategorijama navedenim u trećem poglavlju. Korisniku se prikazuje špil karata, na svakoj karti nalazi se po jedan graf (slika 4.4). Povlačenjem karte s vrha špila prema jednom od 4 kuta zaslona mobilnog uređaja, dolazi do sortiranja karte u određenu kategoriju. Područja sortiranja su sljedeća: povlačenje u

gornji lijevi kut dodaje kartu u kategoriju *repeat* preklapanja, gornji desni u kategoriju kimernih, donji lijevi u kategoriju regularnih te donji desni kut u kategoriju *low quality* preklapanja.



Slika 4.4: Izgled korisničkog sučelja za sortiranje grafova

Špil karata predstavljen je razredom `CardStack` iz biblioteke SwipeableCardStack. Ovaj razred upravlja općim postavkama poput veličine špila i rotacije karata pri povlačenju, a također vodi računa i o `SortingStackAdapter` adapteru za prikaz pojedine karte. Osim toga, pridružuje mu se i instanca sučelja `CardStackListener` koje osluškuje aktivnosti nad špilom te na temelju prikupljenih informacija pokreće pripadajuće metode za obradu.

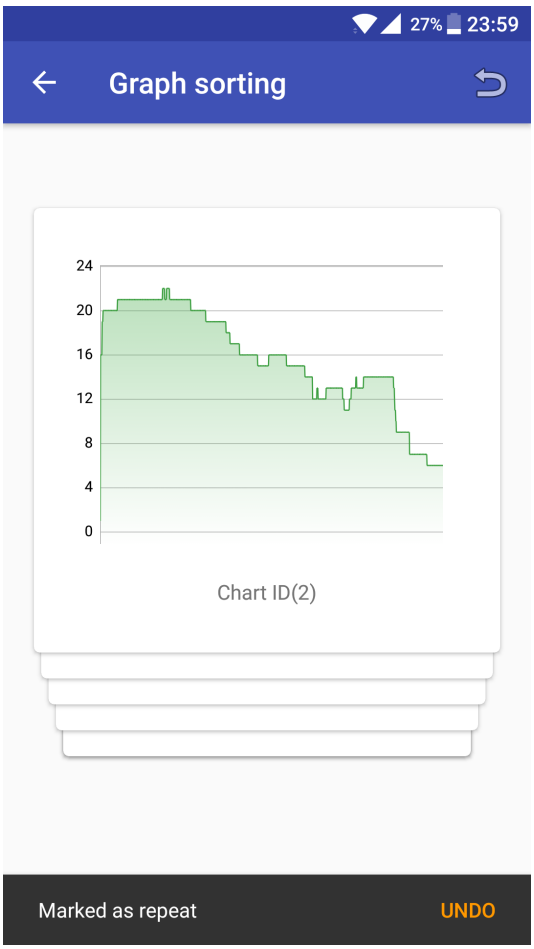
Svaka karta u špilu temeljena je na razredu `CardView` iz Android Support Library-a[18]. XML definicija karte sadrži vertikalnu grupu u kojoj se nalazi prikaz grafa elementom `<LineChart>`, dok se ispod njega nalazi `<TextView>` element koji sadrži naslov

[18] Android Support Library – Skup manjih biblioteka koje sadrže često korištene razrede i elemente ne uključene u Android SDK-u

grafa. Za prikaz pojedine karte zadužen je razred `CardStackAdapter`. Tijekom učitavanja karata, adapter za svaku od njih konstruira prazan okvir te ga puni sadržajem. Po ispunjavanju karte, adapter ju dodjeljuje roditeljskoj grupi pogleda (engl. *view group*), koja ju iscrtava na zaslon kada to bude potrebno.

Unutar špila nalazi se maksimalno 6 karata u bilo kojem trenutku. U trenutku kada broj karata u špilu padne ispod 4, pokreće se asinkroni zadatak koji generira nove karte s podatcima prikupljenim iz baze podataka te ih postavlja na dno špila. Ukoliko se dogodi da korisnik izvrši neželjenu radnju nad špilom, metoda `CardStack#undo()` nudi mogućnost poništavanja posljednje radnje i povratka špila u prethodno stanje.

Nakon povlačenja jedne karte, aktivira se metoda vezana uz pripadajuće područje zaslona. Sortiranom `RealmChartModel` objektu dodjeljuje se kategorija, a ta informacija zapisuje se u bazu putem metode `Realm#copyToRealmOrUpdate(...)`. Kako bi se korisnika obavijestilo o uspješnosti operacije, na dnu zaslona pojavljuje se *snackbar*[19] poruka s informacijom o odabranoj kategoriji (slika 4.5).



Slika 4.5: Prikaz *snackbar* poruke

[19]Snackbar – Android komponenta koja sadrži redak teksta i tekstualni akcijski gumb

4.3.3. Završna obrada i zapisivanje

Rezultati sortiranja i označavanja pohranjeni su na lokalnom uređaju. Kako bi se oni mogli obraditi i iskoristiti, potrebno ih je poslati udaljenom poslužitelju, putem kojeg će im se moći jednostavno pristupiti putem HTTP zahtjeva.

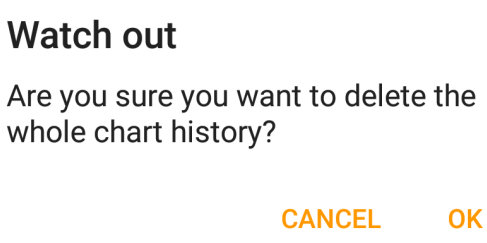
Slanje rezultata vrši se u obliku slanja datoteke koju je potrebno izgenerirati koristeći informacije pohranjene u bazi podataka. Generiranjem upita prema bazi, dohvaćaju se oni rezultati koji još nisu bili poslani na poslužitelj. Prije no što se započne s prebacivanjem podataka u tekstualnu datoteku, važno je prisjetiti se kako su grafovi na mobilnim uređajima iscrtavani na smanjenom skupu točaka, iz razloga objašnjenih početkom prethodnog potpoglavlja. U skladu s tim, potrebno je X koordinate svih početnih i završnih točaka označenih regija translatirati udesno.

Kako je upis u tekstualnu datoteku dugotrajna i blokirajuća izlazna operacija, potrebno ju je izvršiti putem asinkronog zadatka. Tijek zadatka korisniku se dojavljuje u obliku trajne notifikacije poslane kroz Android sustav.

Zapisi koji prikazuju označavanje regija sadrže naslov grafa kojeg slijede 2 uređena para koji predstavljaju početnu i završnu točku regije (npr. *Chart ID(1) (2562, 35) (5736, 41)*), dok zapisi kojima je prikazana kategorija sortiranja grafova sadrže naslov grafa i kategoriju sortiranja (npr. *Chart ID(1) Repeat*).

Završetak pisanja datoteke pokreće operaciju generiranja *multipart*[20] `HTTP POST` zahtjeva. Ukoliko je zahtjev zaprimljen i uspješno obrađen na strani poslužitelja, korisnika se kroz trenutno aktivnu trajnu notifikaciju obaviještava o završetku slanja.

Kada se svi pripremljeni grafovi obrade, a rezultati pošalju na poslužitelj, moguće je obrisati podatke s uređaja, kako bi aplikacija bila spremna za novi skup grafova. Kako je ovakva operacija nepovratna, prije brisanja podataka korisniku se prikazuje dodatan sigurnosni dijaloški okvir kojeg mora potvrditi ukoliko zaista želi obrisati sve podatke (slika 4.6).



Slika 4.6: Dijaloški okvir za potvrdu brisanja svih podataka

[20]Multipart zahtjev – HTTP zahtjev namijenjen slanju jedne ili više datoteka sadržanih u tijelu

5. Ispitivanje i performanse

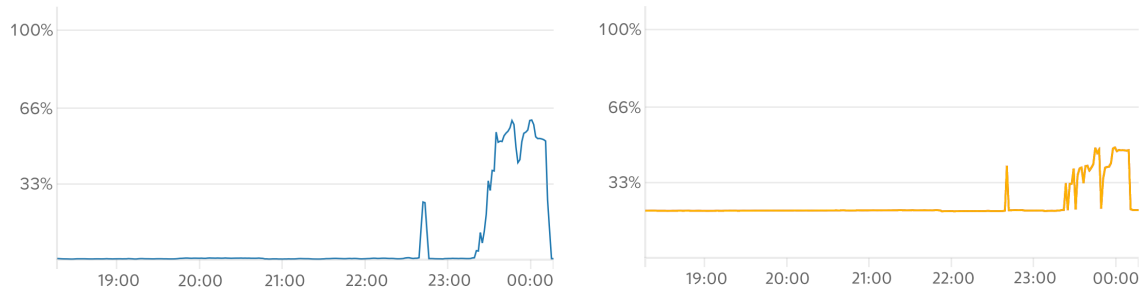
Testni uređaj za ispitivanje mobilne aplikacije bio je OnePlus 2 s 8-jezgrenim Qualcomm Snapdragon 810 procesorom, Adreno 430 grafičkim procesorom te 4 GB radne memorije. Poslužitelj na kojem je pokrenuta web aplikacija temeljen je na dva 8-jezgrena Intel® Xeon® E5-2650L procesora uz 2 GB radne memorije.

Ispitivanjem je provjeravano vrijeme potrebno za procesiranje zahtjeva s PAF datotekom na poslužiteljskoj strani te preuzete datoteke unutar mobilne aplikacije.

5.1. Rezultati ispitivanja

5.1.1. Poslužitelj

Ispitivanje poslužitelja izvršeno je slanjem zahtjeva koji su sadržavali PAF datoteke od 10.000, 20.000, 50.000, 100.000 i 300.000 redaka. Testne datoteke temeljene su na PAF datoteci dobivenoj sekvenciranjem bakterije *Escherichia coli*, koristeći sekvencer tvrtke *Oxford Nanopore Technologies*[21]. Iskorištenje procesora i radne memorije prikazano je sljedećim grafovima. Slanje zahtjeva vršeno je iterativno od najmanje datoteke, pa sve do najveće.



(a) Graf iskorištenja procesora **(b)** Graf iskorištenja radne memorije

Slika 5.1: Prikaz iskorištenja resursa na poslužitelja tijekom testiranja

Analizom grafa iskorištenja procesora, primjetno je da trenutni model programa za parsiranje ne koristi sve mogućnosti koje poslužiteljski procesori nude.

[21] https://nanoporetech.com

Prikaz vremena obrade datoteka na poslužitelju dan je u tablici 5.1.

Datoteka	Broj redaka	Vrijeme obrade (s)
10k.paf	10.000	22
20k.paf	20.000	45
50k.paf	50.000	111
100k.paf	100.000	265
300k.paf	300.000	771

Tablica 5.1: Vrijeme obrade PAF datoteka na poslužitelju

Pregledom rezultata, vidljivo je da vrijeme obrade raste linearno s porastom broja redaka. Razlog tome je korištenje samo jedne linije izvođenja pri radu programa za parsiranje.

5.1.2. Mobilna aplikacija

Kako rezultati obrade prethodno korištenih datoteka sadrže iznimno velik broj grafova, njihovo korištenje na mobilnim uređajima je nepraktično. Stoga su u procesu ispitivanja rada mobilne aplikacije korištene datoteke s 50, 100, 150 i 200 grafova koje je potrebno pripremiti za korištenje. Kako bi se umanjio utjecaj različitosti grafova na ispitivanje, za svaku kategoriju kreirane su 3 datoteke s različitim grafovima, a potom je izračunata aritmetička sredina njihovih vremena obrade.

Rezultati ispitivanja prikazani u tablici 5.2 pokazuju da je omjer broja grafova i vremena obrade najbolji za datoteke koje sadrže između 100 i 150 grafova. Ova informacija korisna je za podešavanje poslužiteljskog programa za parsiranje, kako bi konstruirao izlazne datoteke optimalne veličine.

Datoteke	Broj grafova	Prosječno vrijeme obrade (s)
Charts50-1.txt Charts50-2.txt Charts50-3.txt	50	22
Charts100-1.txt Charts100-2.txt Charts100-3.txt	100	52
Charts150-1.txt Charts150-2.txt Charts150-3.txt	150	134
Charts200-1.txt Charts200-2.txt Charts200-3.txt	200	221

Tablica 5.2: Vrijeme obrade datoteka na mobilnom uređaju

6. Zaključak

Kroz ovaj rad razvijen je i predstavljen sustav za ručno otkrivanje lažnih preklapanja koji nastaju prilikom procesa sastavljanja genoma. Potreba za kreiranje ovakvog sustava pokazala se pri razvoju algoritma strojnog učenja kojim se vrši sastavljanje genoma, a koji zahtijeva velike količine raznovrsnih podataka za treniranje i provjeru. Upravo će se rezultati dobiveni putem ovog sustava koristiti kao temelj za generiranje tih podataka.

Sustav čine mobilna Android aplikacija te središnji udaljeni poslužitelj. Mobilne aplikacije nude najbolju korisničku interakciju za ovakvu vrstu obrade podataka, dok Android platforma pruža pristup najvećem broju krajnjih korisnika. Kako bi se smanjilo opterećenje mobilnih uređaja koje nastaje prilikom obrade PAF datoteka, takve operacije prebačene su na udaljeni poslužitelj, putem kojeg se također pojednostavljuje i distribucija podataka za obradu prema mobilnim uređajima te prikupljanje i pohrana rezultata s istih.

Glavni izazov pri izradi programskog rješenja bio je kreirati jednostavno i intuitivno korisničko sučelje za označavanje regija grafova preklapanja i sortiranje grafova po kategorijama. Rad po službenim preporukama za razvoj Android aplikacija, izbor klijent – poslužitelj arhitekture te korištenje vanjskih biblioteka za prikaz grafova preklapanja te karata za sortiranje grafova uvelike su olakšali razvoj mobilne aplikacije.

Daljnji razvoj temeljit će se na problemima ukazanim pri ispitivanju sustava. Rezultati ispitivanja pokazuju da postoji prostor za dodatno ubrzanje obrade podataka te poboljšanje performansi kako aplikacije, tako i poslužitelja. S poslužiteljske strane, najveći napredak u brzini rada može se postići paralelizacijom programa za parsiranje PAF datoteka. Poboljšanje performansi mobilne aplikacije temelji se na smanjenju dvostrukih iscrtavanja područja zaslona, optimizaciji hijerarhije pogleda, odgađanju iscrtavanja pozadinskih objekata te paralelizaciji ključnih aktivnosti i zadataka.

Koncentracijom na poboljšanje performansi na klijentskoj strani uklonit će se i restrikcije zbog kojih trenutno nije moguće koristiti potpuni skup točaka dostupan

za svaki graf. Time će doći i do povećanja preciznosti označavanja regija grafova koje predstavljaju kimerna i ponavljajuća preklapanja, što dovodi i do kvalitetnijih konačnih rezultata.

Literatura

Šikić, M., Domazet-Lošo, M. (2013). Bioinformatika.
http://www.fer.unizg.hr/predmet/bio *(2017-05-20)*

Li, H. (2016). Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*

Langmead, B. (2014). Lecture notes. Johns Hopkins University, Whiting School of Engineering. http://www.langmead-lab.org/teaching-materials *(2017-05-21)*

Treangen, T., Salzberg, S. (2012). Repetitive DNA and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, 13.
http://www.nature.com/nrg/journal/v13/n1/full/nrg3117.html *(2017-05-21)*

Ekblom, R., Wolf, J. (2004). *A field guide to whole-genome sequencing, assembly and annotation.* Evolutionary Applications.
http://onlinelibrary.wiley.com/doi/10.1111/eva.12178/full *(2017-05-23)*

National Center for Biotechnology Information Assembly Basics (2010).
https://www.ncbi.nlm.nih.gov/assembly/basics *(2017-05-25)*

Network Working Group, RFC 4122 *(2017-05-27)*

Android Developer API Guides, https://developer.android.com/index.html *(2017-05-27)*

About Node.js® (2017). https://nodejs.org/en/about *(2017-06-01)*

OkHttp overview & documentation (2017). http://square.github.io/okhttp *(2017-06-03)*

Best Practices for Android Performance (2017).
https://developer.android.com/training/best-performance.html *(2017-06-04)*

Ručno određivanje lažnih preklapanja koja nastaju pri sastavljanju genoma

Sažetak

Proces sastavljanja genoma temeljen je na analizi preklapanja kratkih očitanja genetskog materijala. Zbog očitanja loše kvalitete, kao i zbog prisutnosti kimernih i ponavljajućih očitanja, osim ispravnih preklapanja pojavljuju se i lažna preklapanja. Detekcija lažnih preklapanja ključna je u postizanju boljeg konačnog rezultata. Jedna od metoda detekcije je generiranje 1D-signala temeljenog na izračunu broja preklapanja svake nukleinske baze pojedinog očitanja, koji se potom prikazuje u obliku grafa. Kroz ovaj rad predstavljeno je aplikacijsko rješenje za Android uređaje, kojim je moguće označiti regije u kojima se pojavljuju lažna preklapanja, kao i sortirati grafove po predefiniranim kategorijama. U radu je data teorijska osnova rješenja, kao i detalji njegove implementacije i konačnog ispitivanja.

Ključne riječi: Lažna preklapanja, kimerna očitanja, ponavljajuća očitanja, sastavljanje genoma, Android, bioinformatika

Manual detection of false overlaps occurring during genome assembly

Abstract

The genome assembly process is based on overlapping and analyzing short reads of genetic information. False positive overlaps arise in addition to true positive overlaps, as a result of low quality, chimeric and repeating reads. Detection of these false positive overlaps is key to constructing more accurate and complete genomes. One of the methods for detecting such overlaps is by generating a 1D-signal based on the number of overlaps of each nucleobase in a read, which can then be shown in the form of a chart. This thesis presents an Android application for demarking areas of false overlaps on the aforementioned charts, as well as sorting charts in predefined categories. Both the theoretical and practical solution, as well as the testing process, have been documented inside of this paper.

Keywords: False overlaps, chimeric reads, repeating reads, genome assembly, Android, bioinformatics